Project Report

(Software Testing:CS 731)

George T. Abraham(IMT2017019)
Sushranth Hebbar(IMT2017042)

Problem Statement:Design Integration Testing for an application

1. Application:- The choice of application(code) for this project is a Horse Racing Record Management System. It performs as a sort of a personal Database which is then used with the intention of storing data pertaining to horse races. With this application, it is easy to keep track of horse races. Attributes like name, finish time, age, weight, etc for each horse can be saved as a record using a Personal Data Store Implementation (PDS). A Binary Search Tree is used to handle the below operations in an efficient manner.

2. Structure of Code:-

The code is structured in the following way; the user is provided with an interactable terminal which is implemented through **“application.c”** which allows him/her to use the basic database functionalities (that is implemented in **“pds.c”**).The interaction with “application.c” calls another module called **“HorseRecord.c”** that in turn calls “pds.c”. The primary use of “HorseRecord.c” is to map the requests/functionality demanded by user to one of the database operations from the “pds.c” module and providing necessary additional internal information(such as unique id/key value,storing the location of a particular entry/horse in the database file) to invoke the PDS functions. The **“bst.c”** module is used to make iteration through the data entries stored in a file faster. This is done by assigning a unique key to every data entry and adding it to a bst (Binary Search Tree).

3. Functionalities supported :-

a. OPEN:- The pds contains a data file and index file. The data file contains information about records in a binary format. Each record has a unique primary key which is used for identification. The index file stores this primary key and also the location of a record in the data file.

b. STORE:- This operation inserts a record into the datafile. It is inserted at the end of the datafile. The primary key of this record and the location of the record in the file are inserted into the BST in the form of a struct. The primary key is used as the key for the BST. In case of duplicates or missing records, it signals an error message.

c. SEARCH_BY_PRIMARY_KEY:- This operation searches for the entry and then returns the record which matches with the primary key. It first searches the primary key in the BST. It retrieves the file offset information from the BST node. The file offset is used to point to the record in the datafile. It is read from that location and a success flag is returned. Otherwise, an appropriate error message is returned.

d. SEARCH_BY_NON_PRIMARY_KEY:- The search key here is any attribute apart from the primary key of the record. As an example,it could be the name of the horse. This operation does a full table scan of the data file until the desired record is found.

e. MODIFY_RECORD_BY_PRIMARY_KEY:- Search for the key in the BST. Seek to the file location based on the offset in the BST node. Write the key and the modified record value at the pointed location of the datafile.

f. DELETE_RECORD_BY_PRIMARY_KEY:- Search for the primary key in the BST. If it is not found, then return an error message. Otherwise, mark it as deleted. When closing the file, write node values from the BST into the index file only if it's deleted flag value is not true.

g. CLOSE:- The contents of the BST have to be saved into the index file. Then the memory used by the BST is freed. Finally, the index file and the data file are closed.

4. Testing Strategy:-

The above application is implemented using C . There are 4 important files which contain the necessary functionalities. They are:

bst.c, pds.c, HorseRecord.c and application.c

The operations described above are implemented in pds.c . They are implemented as separate methods. Internally, they invoke methods related to bst which are implemented in bst.c . HorseRecord.c contains methods which act as a wrapper on top of pds.c . It contains record related details. Application.c acts as a user interface.

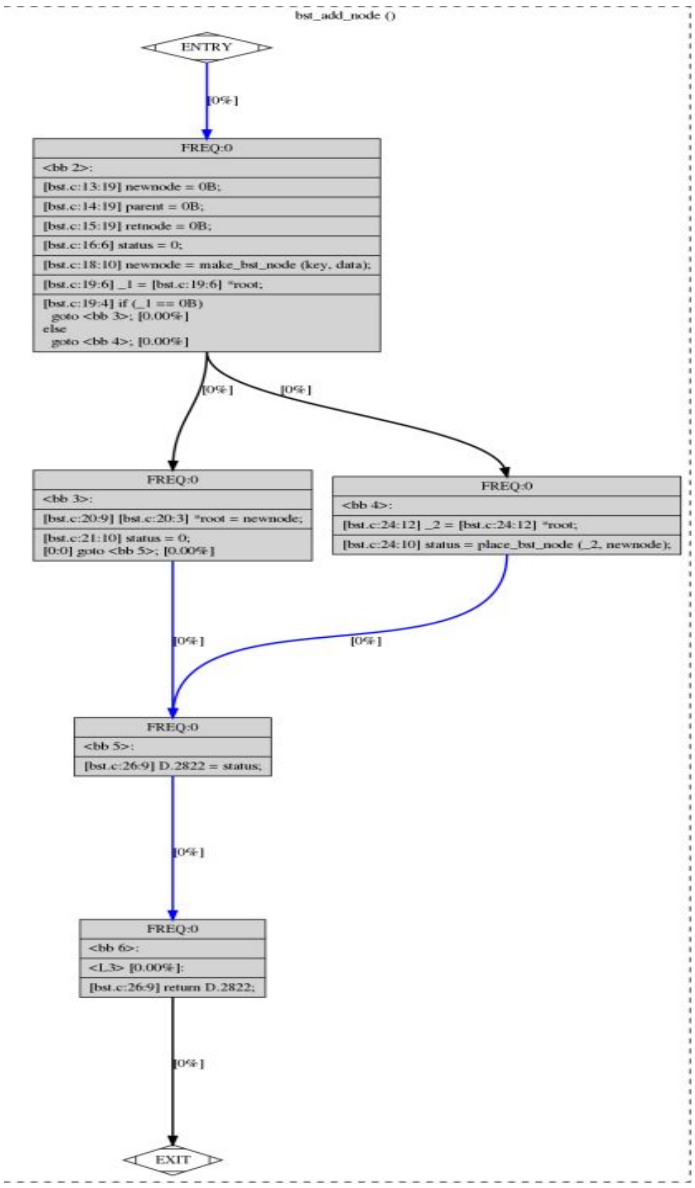
The problem that we have selected is **Design-Integration-Graphs** . So, integration testing is crucial. The designed test cases use criteria based on last defs and first uses. **Coupling-All-Use-Coverage criterion** was used to design test cases. A top down integration approach was used and test stubs were used appropriately. So,the approach taken was to handle testing between modules of “Horserecord.c and pds.c” based on the above criteria(interactions/ function calls between the two were analyzed) . Then followed by tests between “pds.c and bst.c” . During our analysis, we observed that some of the coupling du-pairs are transitive in nature, i.e, these pairs were passed directly as arguments from one function to another. In order to satisfy the definition of last-def and first-use, we defined these variables at the start of each module. Now the definition is satisfied and it is possible to analyze the data flow across consecutive interfaces.

(Note:*The interaction between “application.c and Horserecord.c” was not considered since application.c was just there as an interactable interface to allow user to perform certain actions . Therefore it was just a series of mappings between user input and the corresponding invocation to the implemented functions in “Horserecord.c” and did not provide much scope for analysis.)*

5. Workflow:-

5.1 Generate Control Flow Graph :- The gcc compiler has the functionality to generate CFG’s per module. The command is **gcc -fdump-tree-cfg-lineno-graph <input_files>**. The output of this are dot extension files. **Graphviz** is an open source library which converts the dot extension files into png files.

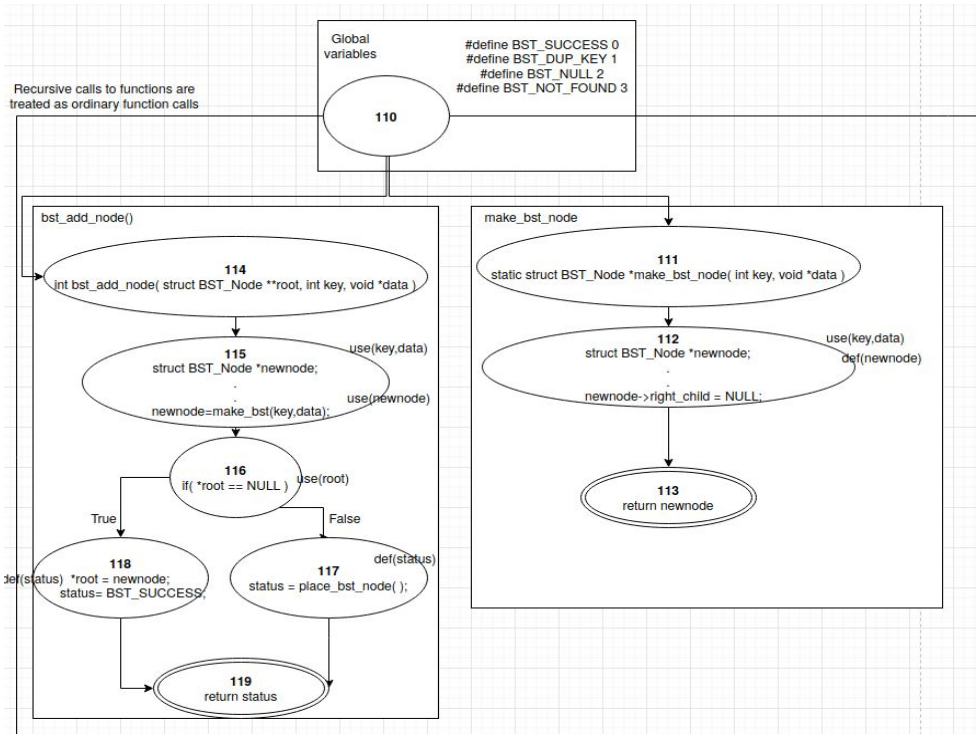
As an example consider the below image which contains the control flow graph for the insert() function in **bst.c** .



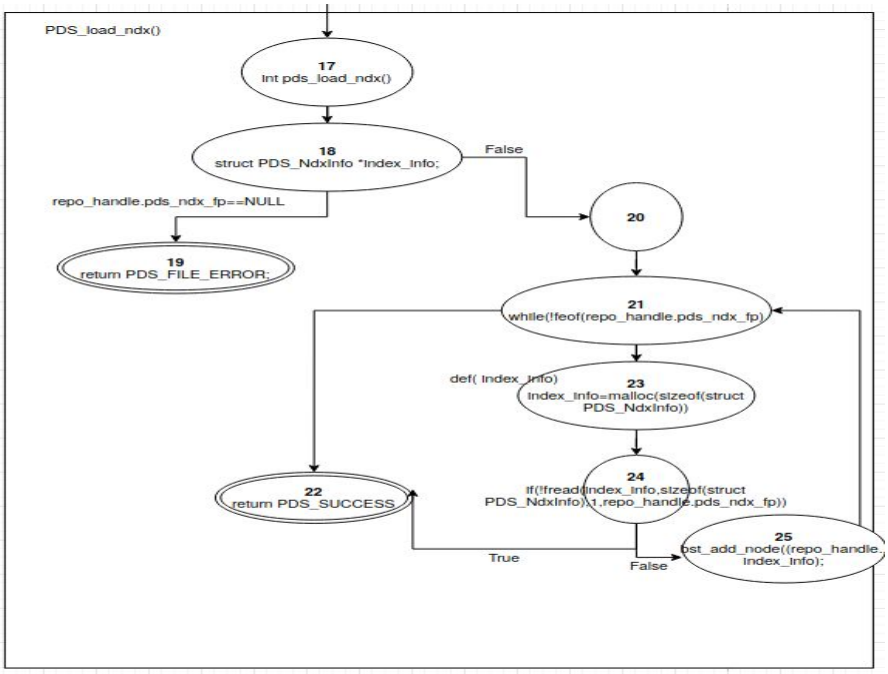
It is important to note that the gcc compiler treats recursive functions as another function call . It uses backward arrows only for handling iterative loops.

As we can see,the CFG generated **does not** label the nodes or mention the **variables used/defined at a particular node** in the CFG.Since the next step (of finding last def and first use pairs) in our analysis required us to identify the def() and use() at nodes in the generated CFGs, this task had to be **done manually**.

By referencing the CFGs automatically created by gcc for each of the modules (bst.c, pds.c, HorseRecord.c and application.c), we annotated definitions,usage at the nodes appropriately for the CFGs created and labelled the nodes.Additionally,we also had to add a super node for every module created to handle the definition of global variables.The super node has an edge to the starting node of every function and solely contains **def()** statements of global variables.Each node is assigned a unique number to make working across modules easy.



*(Figure 1.Example of CFG with marked defs and uses in **bst.c** . Node **110** is the supernode responsible for declaring global variables from **bst.c** and we can see that **110** has edge to all the CFGs for every function in **bst.c** so that we can create accurate last-def and first use for global variables too. Individual sections of CFGs for functions are also mapped out.Ex:bst_add_node() and make_bst_node().The parameter " **root" passed to bst_add_node() is used at node **116**,similarly the parameters "key,data" are used at node **115**).*

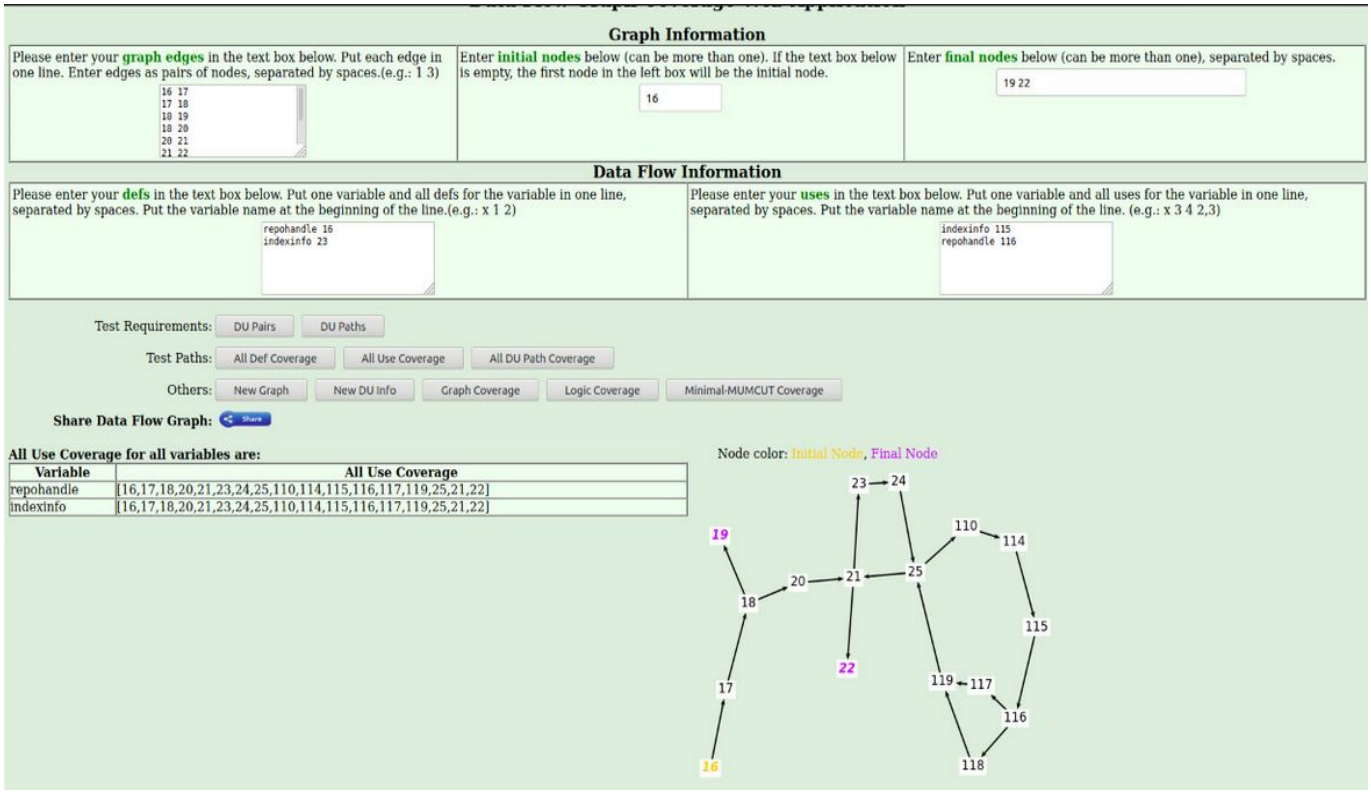


*(Figure 2.CFG from **pds.h,** the graph here corresponds to the function pds_load_ndx())*

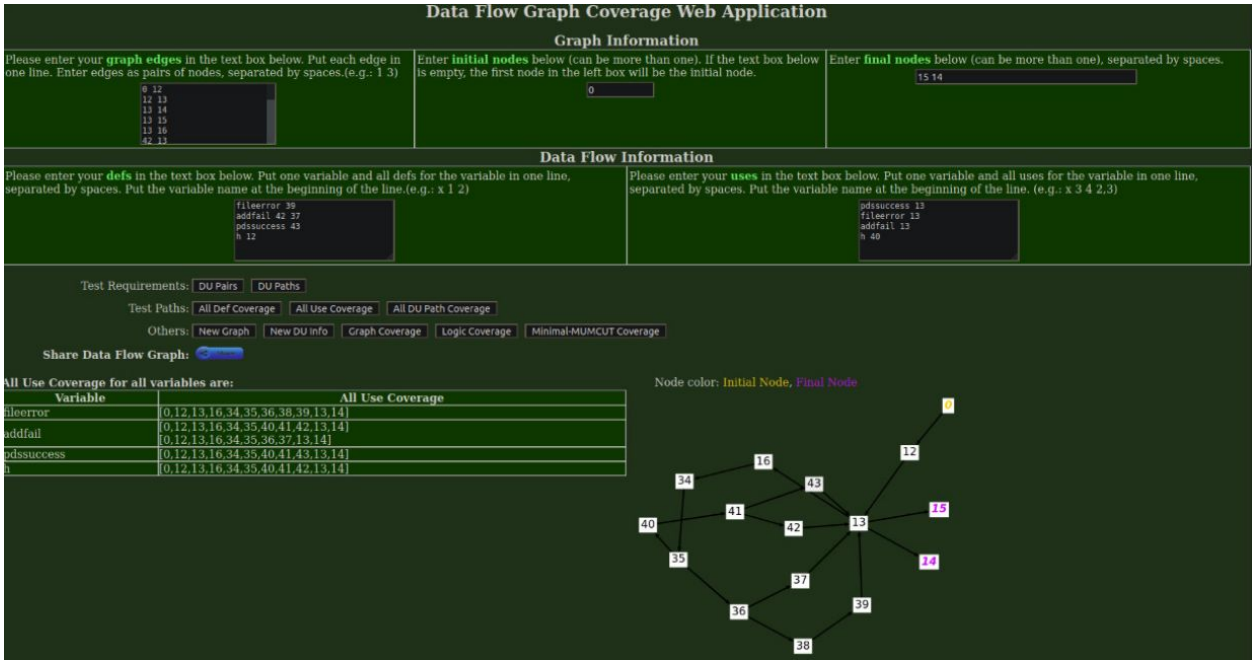
After manually annotating def and use sites in a CFG for various functions,we use them for last-def and first-use analysis across modules.

***(NOTE:-** This is saved in the form of .io extension files inside the folder **annotated_cfg**. To view them, open https://app.diagrams.net/ and then click on upload to view the chosen CFG for that module.)*

5.2 Generate Data Flow Graph:- The next step is to create appropriate links between the modules by making them invoke each other. Since,there was no readily available open source software (for the C language) to do integration testing across different functions in different modules automatically, we constructed edges between the **caller and callee** functions across the different modules based on invocation and return values. Then the coupling du pairs are served as input into the webapp (https://cs.gmu.edu:8443/offutt/coverage/DFGraphCoverage).
Then paths are generated with regards **to all use coupling coverage**. Test cases are then designed based on these paths.



*(Figure 3.All use coverage between pds_load_ndx() and bst_add_node(). pds_load_ndx() is a function from **pds.c** whereas bst_add_node() is from **bst.c** .The graph is constructed by referencing the annotated CFGs from each module.The edges and nodes input are directly from the same after adding additional nodes between caller and callee.)*

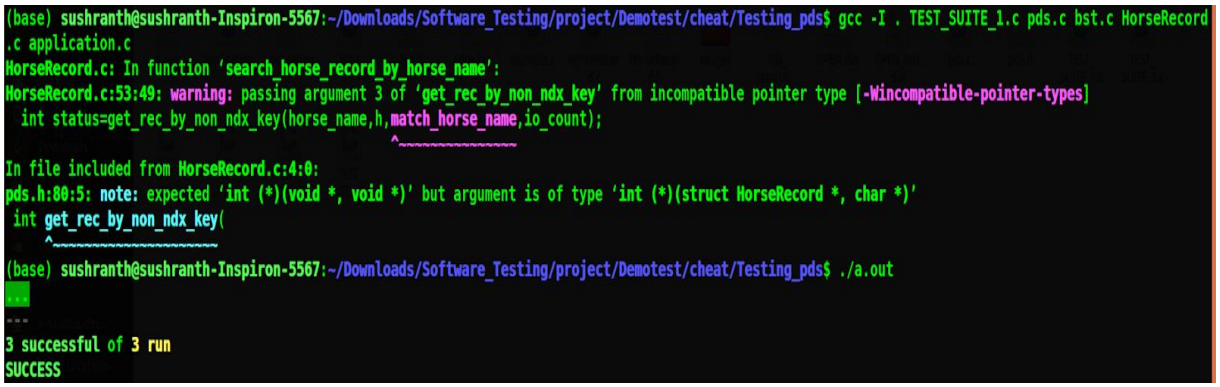


*(Figure 4.All use coverage between add_horse_record_by_competition_id()[from **Horserecord.c**] and put_rec_by_ndx_key()[from **pds.c**])*

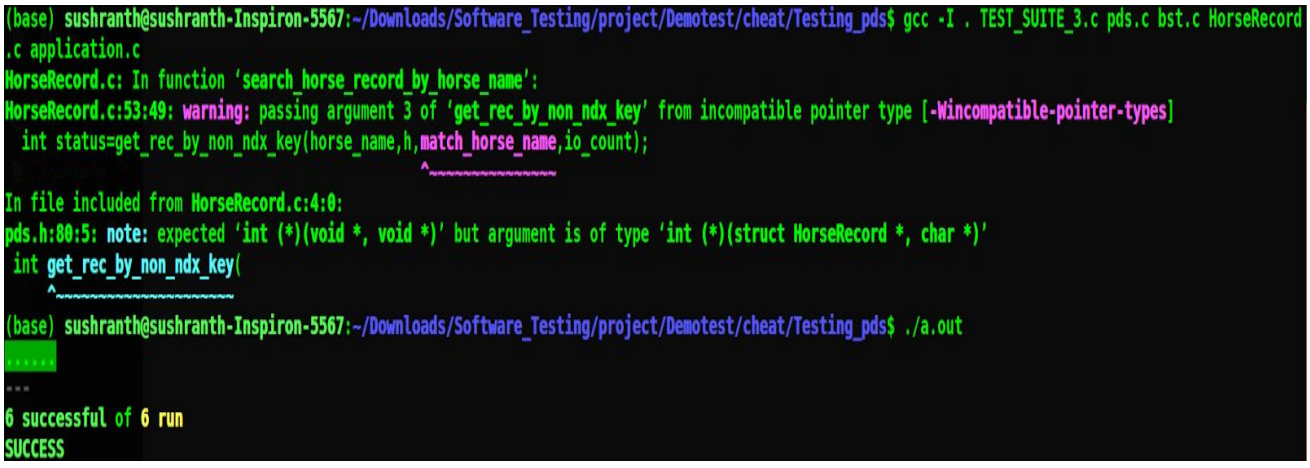
5.3 Test Case Design:- The C Header Automated Testing (CHEAT) is an open source testing framework designed for the C programming language. (https://github.com/Tuplanolla/cheat). Test cases have been divided into several test suites where each suite corresponds to a set of all-use-coupling paths between two specific modules across two different files. The test suites can be found in the folde**r TEST_SUITE.**

gcc -I . TEST_SUITE_i.c pds.c bst.c HorseRecord.c application.c
Compile the ith test suite with the above command and run the executable.

As an example, the test suite for figure 3 can be found in the file TEST_SUITE_1.c.



Similarly for figure 4, the file TEST_SUITE_3.c contains the test cases.



Here is an example of sample code from a test_suite.

```
CHEAT_TEST(NON_KEY_SEARCH,

  char horse_name[50];
  strcpy(horse_name, "Name-of-horse-1");
  struct HorseRecord test,test1;
  int actual_io = 0;
  int expected_io = 1;
  int competition_id = 1;
  int non_search_status = search_horse_record_by_horse_name( horse_name, &test, &actual_io );
  create_record(&test1,competition_id);
  // printf("%d %d %d\n",non_search_status,check(&test,&test1),actual_io);
  cheat_assert((non_search_status==PDS_FILE_ERROR));
)
```

In total, there are 8 test suites and a total of 35 test cases across all modules.

Individual Contributions:-

- George Abraham:
  Creating annotated CFG for pds.c, Identifying and creating test cases and test suites for integration between the modules Horserecord.c and pds.c
- Sushranth Hebbar:
  Created annotated CFG for HorseRecord.c and bst.c. Identifying and creating test cases and suites for integration between pds.c and bst.c